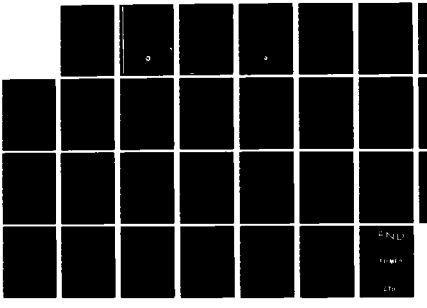
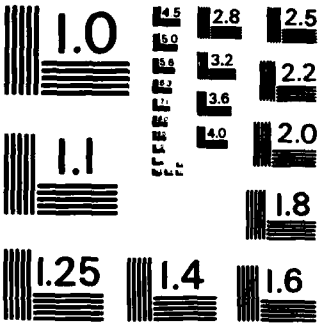
AD-A162 698 GETWRD PACKAGE UPDATE: NEW FEATURES AND MODIFICATIONS TO THE GETWRD PACKAGE(U) DEFENCE RESEARCH ESTABLISHMENT ATLANTIC DARTMOUTH (NOVA SCOTIA) D HALLY AUG 85 1/1

UNCLASSIFIED DREA-TC-85/312 F/G 9/2 NL





UNLIMITED DISTRIBUTION

National Defence
Research and Development Branch

Défense Nationale
Bureau de Recherche et Développment

TECHNICAL COMMUNICATION 85/312
AUGUST 1985

GETWRD PACKAGE UPDATE: NEW FEATURES AND MODIFICATIONS TO THE GETWRD PACKAGE

David Hally

Defence Research Establishment Atlantic

Centre de Recherches pour la Défense Atlantique

Canada

DEFENCE RESEARCH ESTABLISHMENT ATLANTIC

9 GROVE STREET
P.O. BOX 1012
DARTMOUTH, N.S.
B2Y 3Z7

TELEPHONE
(902) 426-3100

CENTRE DE RECHERCHES POUR LA DÉFENSE ATLANTIQUE

9 GROVE STREET
C.P. 1012
DARTMOUTH, N.É.
B2Y 3Z7

UNLIMITED DISTRIBUTION

National Defence
Research and Development Branch

Défense Nationale
Bureau de Recherche et Développment

GETWRD PACKAGE UPDATE: NEW FEATURES AND MODIFICATIONS TO THE GETWRD PACKAGE

David Hally

AUGUST 1985

Approved by T. Garrett Director/Technology Division

DISTRIBUTION APPROVED BY

D/TD

TECHNICAL COMMUNICATION 85/312

Defence Research Establishment Atlantic

Centre de Recherches pour la Défense Atlantique

Canada

Abstract

The GETWRD Package is a library of procedures designed to ease the implementation of command languages using FORTRAN 77. It allows the user to interpret a *word* of user input by matching it with one of the entries in a user-supplied dictionary. Features designed to increase the friendliness of the program/user interface include a type ahead facility, recognition of abbreviations, word completion and understandable error messages.

This memorandum describes enhancements to the GETWRD Package. While most of the enhancements are of most benefit to the programmer, a major improvement to the program/user interface is the inclusion of a spelling corrector which will catch most typographical errors committed by the user. Other improvements include the ability to interpret the *word* of input as a simple string with no dictionary matching, a logical variable, the answer to a Yes-No question, or a number in a specified range; greater flexibility for the programmer in formatting prompt and help messages; and a sorting routine which can be used to ensure that the dictionary is in alphabetic order.

All changes have been implemented to be upwardly compatible with the original version of the GETWRD Package so that no changes need be made to existing code which calls GETWRD Package procedures.

Résumé

Le progiciel GETWRD est une bibliothèque de procédures destinées à faciliter la mise en oeuvre de langages de commande faisant appel au FORTRAN 77. Ce progiciel permet à l'utilisateur d'interpréter un mot d'une entrée de données en appariant ce mot à un de ceux qui sont contenus dans un dictionnaire fourni par l'utilisateur. Diverses fonctions - commande anticipée, reconnaissance des abréviations, capacité de compléter les mots et messages d'erreurs compréhensibles - accroissent la facilité d'utilisation du système.

Le présent document décrit les perfectionnements apportés au progiciel GETWRD. Bien que la plupart de ces changements concernent le programmeur, l'usager bénéficie aussi d'une amélioration importante; en effet, un dispositif de correction des fautes d'orthographe permet désormais de supprimer la majorité des erreurs typographiques commises par l'usager. Parmi les autres nouveautés, citons aussi la capacité d'interpréter un mot d'introduction comme une simple suite de caractères sans l'apparier aux mots du dictionnaire et d'interpréter également une variable logique, la réponse (oui-non) à une question ainsi qu'un nombre dans une fourchette donnée. Le système offre aussi au programmeur plus de souplesse dans la mise en forme des messages guide-opérateur et des messages SOS et un programme de tri qui permet de s'assurer que le dictionnaire est en ordre alphabétique.

Tous les changements introduits sont à compatibilité ascendante avec le progiciel GETWRD original, de sorte qu'il n'est pas nécessaire de modifier le code d'appel actuel des procédures du progiciel.

A-1

Table of Contents

1 INTRODUCTION

The GETWRD Package[1] is a library of procedures designed to ease the implementation of command languages written in FORTRAN 77. The procedures provide a friendly program/user interface with very little effort by the programmer. Among the features of the GETWRD Package in its original version were a type ahead facility, command completion, recognition of abbreviated commands, the ability to enquire about possible valid input, and controlled aborts (i.e.the program will abort only when the user requires it to). Several procedures have been added to the original version of the GETWRD Package and several of the existing subroutines have been enhanced. All modifications are completely compatible with any existing code which uses the original package so that no changes to existing programs need be made to use the updated versions of the procedures. Most of the modifications will be noticed only by programmers using the GETWRD Package. They are offered several new features which will widen the applicability of the package and its ease of use. These are the ability to have complicated formats for prompts and help messages, a tagged sorting subroutine which will sort the dictionary into alphabetical order, the ability to input character strings that will not be matched with dictionary entries, a procedure for returning a yes-no answer, a procedure for returning the value of a logical variable, and the ability to input numbers which are restricted to a specified range. However, there is one major new feature that will be noticed by users of programs in which the GETWRD Package is invoked: a spelling correction facility has been added that will catch many typographical errors and much reduce time wasted re-entering commands due to slips of the finger.

2 USE OF THE NEW FEATURES

In the following sections the use of the new features is described in detail. The manner in which they have been implemented is described in Appendix A.

2.1 FORMATTING ENHANCEMENTS

One of the limitations of the original version of the GETWRD Package was the lack of control the programmer had over the format of the prompt and help messages. These messages were passed to the GETWRD subroutines in the character variables PROMPT and HLPMSG respectively, and written, when needed, on a single line of the terminal screen. If the help message was invoked, it was automatically followed by the list of allowed commands, one command per line following the help message. By setting flags in the common block / FMTFLG / in the new version, the programmer may create messages having much more complicated formats. The programmer is given the option of having the variables PROMPT and HLPMSG interpreted as a simple string as before, or as a format specifier which when used in a WRITE statement will result in a formatted message. The latter option, while slightly more complicated to use, allows the programmer much greater freedom in the appearance of the messages. The form of the WRITE statement used is simply,

```
WRITE(UNITNO,HLPMSG)
```

or

```
WRITE(UNITNO,PROMPT)
```

In the latter case, if the buffer is not empty, the writing of the prompt is immediately followed by the writing of the buffer contents up to the current position:

```
WRITE(UNITNO,'('',+'',A$)')BUFFER(:BUFPOS)
```

The '+' in the format string suppresses the line-feed normally inserted before BUFFER is written: thus, BUFFER will always be written on the same line as the last line of the prompting message. Thus, the prompt format specifier in PROMPT, should always contain a slash, /, to start a new line, or a dollar sign, $, to suppress the carriage return normally inserted at the end of the execution of a WRITE statement. Otherwise the contents of the buffer will be written at the start of the last line of the prompt, overwriting whatever has been written there. Note that a dollar sign as a carriage return suppressant is a widely used but non-ANSI convention which may not be allowed on some computers (see Appendix E, Reference 1). As an example, suppose the following prompt spanning three lines were to appear on the terminal screen:

```
Enter a new number between 1 and 10:

   =>NEW-NUMBER
```

where 'NEW-NUMBER' is what is currently in the buffer. This may be achieved by setting PROMPT to the following format string:

```
PROMPT = '('' Enter a new number between 1 and 10:''//5X,''=>''$)'
```

The dollar sign at the end of the prompt suppresses the carriage return that would normally be inserted after the WRITE statement so that when the buffer contents are written after the prompt they will appear to the right of the arrow, '=>'. If the $ was omitted the prompt would appear as

```
Enter a new number between 1 and 10:
NEW-NUMBER
```

the buffer contents having overwritten the arrow. If a slash were used instead of a dollar sign, the buffer contents would appear on the following line:

```
Enter a new number between 1 and 10:

   =>
NEW-NUMBER
```

Note that within the format string, single quotes must be replaced by two single quotes in accordance with the FORTRAN 77 standard for representing single quotes in character strings. Unfortunately, this often makes the format statements difficult to read. To avoid this difficulty, the programmer is allowed the option of replacing all single quotes within the format string with double quotes. In the example above, PROMPT could be set to

```
PROMPT = '(" Enter a new number between 1 and 10:"//5X,"=>"$)'
```

Within the GETWRD Package the double quotes are replaced by single quotes before the string is used in the WRITE statement. In order to allow the natural use of double quotes, the replacement of single by double quotes is only an option which may be turned on by setting a flag in the common block / FMTFLG /.

The programmer is also allowed greater freedom in the way that the list of available commands is written following the help message. The list may be suppressed altogether. This is often appropriate when the dictionary is very short and the options can be included in the help message itself. For example, if the dictionary consists only of the two words 'NO' and 'YES', an appropriate help message might be

```
Enter YES or NO.
```

and the listing of the allowed commands is then superfluous. When the dictionary is large, the listing of one allowed command per line often means that the full dictionary cannot appear on a single terminal screen. For this reason the programmer is now allowed the option of specifying the number of allowed commands to be printed per line. The default is one per line in keeping with the original version of the GETWRD Package. Thus, by specifying the number of allowed commands per line to be 3, a help message which formerly appeared as

```
Enter one of the following commands
ABORT
CEASE
DESIST
DIE
EXIT
GIVE-UP
STOP
SURRENDER
WITHER
```

would appear as

```
Enter one of the following commands
ABORT       CEASE       DESIST
DIE         EXIT        GIVE-UP
STOP        SURRENDER   WITHER
```

The separation between each column of commands is three spaces. It is up to the programmer to ensure that the horizontal extent of the commands will not overflow the terminal screen.

The formatting options described above are invoked by the programmer by setting the values of flags in the common block / FMTFLG /:

```
COMMON / FMTFLG / IPMT, IHLP, IQUOT, NACPL
```

where

IPMT = 0, indicates that PROMPT is to be treated as a simple string as in the original version of the GETWRD Package. This is the default.
= 1, indicates that PROMPT is to be treated as a format specifier.

IHLP = 0, indicates that HLPMSG is to be treated as a simple string as in the original version of the GETWRD Package. This is the default.
= 1, indicates that HLPMSG is to be treated as a format specifier.

IQUOT = 0, indicates that internal conversion from double quotes to single quotes will not occur. This is the default.
= 1, indicates that double quotes will be converted to single quotes by the GETWRD procedures.

NACPL = The number of allowed commands per line to be written after the help message. If NACPL is zero, the allowed commands will be suppressed completely. The default value for NACPL is 1.

2.2 CORRECTION OF TYPOGRAPHICAL ERRORS

An annoying feature of the original GETWRD Package is that, if the user makes a mistake on input, then when prompted to correct the mistake all further input stored in the buffer is lost. If the type ahead facility has been used, a lot of typing may be wasted over a simple typographical error. For example, if when prompted the user enters

```
=>MOVE THE SUQ UP 2 AND THE ASTERISK DOWN 4
```

meaning the third word to be 'SQU' rather than 'SUQ' so that it would match the dictionary word 'SQUARE', then the original version of GETWRD would respond

```
?? Word not in dictionary

=>MOVE THE S
```

and await further input. All the remaining words in the buffer would have been lost. The new version of GETWRD realizes that the user probably meant 'SQU' and responds instead

```
?? Word not in dictionary

Did you mean 'SQU' for 'SUQ' to match with 'SQUARE'? (Y or N)
```

If the user responds 'Y', the mistake will be corrected in the buffer, and the correct match returned from GETWRD. The user will not be reprompted and the remainder of the buffer will be processed normally. If the user enters 'N', then there will be a new prompt for further input and the remainder of the buffer will be lost.

The algorithm used to catch typographical errors is implemented in the subroutine CORSPL. It is derived from an algorithm of Durham, Lamb and Saxe[2] and relies on the observation that most typographical errors fall in one of four categories:

1) a single character is missing from the word,

2) an extra character is contained in the word,

3) two adjacent characters are transposed, or

4) a single character has been replaced by an incorrect character.

CORSPL takes the current word in the buffer ('SUQ' in the example above), then checks each dictionary word in turn to see if it could be identified with the buffer word by one of the four transformations above. If the dictionary word can be matched in this sense, the user will be prompted and asked if that is what was meant. If no dictionary words can be matched, then the simple error message will be written and the user reprompted, losing the remainder of the buffer.

2.3 CORRECTION OF AMBIGUOUS WORD COMPLETIONS

Another mistake which often causes loss of user input is when the completion character is used but a unique dictionary word is not identified. For example, suppose when prompted the user enters

```
=>MOVE THE S$ UP 2 AND THE ASTERISK DOWN 4
```

meaning the third word to match with 'SQUARE'. However, in the example, 'STAR' is also a dictionary word. The original version of GETWRD would respond

```
?? Word completion not unique

=>MOVE THE S
```

and await further input. All the remaining words in the buffer would be lost. The new version of GETWRD tries to recover from this error by prompting the user for the correct word:

```
?? Word completion not unique

Did you mean 'S' to match with 'SQUARE'? (Y or N)
```

If the user responds 'Y', the 'S' in the buffer will be substituted by the full word 'SQUARE' and the match returned from GETWRD. The user will not be reprompted and the remainder of the buffer will be processed normally. If the user enters 'N', the next possible match will be tried:

```
Did you mean 'S' to match with 'STAR'? (Y or N)
```

If the user again answers 'N' and no further words in the dictionary begin with S, the user will be reprompted for further input and the remainder of the buffer will be lost. Since the user did not wish any of the allowed completions of the word, the input should begin with either the *Delete Letter Character*, the *Delete Word Character* or the *Abort Character* in order to clear the 'S' from the buffer.

2.4 INPUT OF CHARACTER STRINGS

The original GETWRD Package allowed two forms of input to the buffer: strings, whose *words* were to be matched with dictionary words, or numbers. A drawback of the original library was the inability to allow the user to input other data types such as strings which would not be matched with dictionary words, or logical variables. For example, a command might require the input of data from a user-specified file. An appropriate command sequence might be

```
=>GET-DATA-FROM DATA.DAT
```

where 'DATA.DAT' is to be interpreted as a file name. Clearly it is impossible to have a dictionary containing all possible file names. The subroutine GETSTR has been provided as a means to input character strings which will not be matched with dictionary words. It interprets the next word in the buffer as a string of characters and returns the word in a character variable, STRING. The position in STRING of the last non-delimiter is returned in the variable LENSTR. Prompting and help messages are as in GETNUM as is the presence of an integer error flag, IER. The user will also be warned when the word in the buffer is to long to fit in the character variable STRING. The special *Abort, Completion, Delete Letter, Delete Word*, and *Help characters* all retain their special functions, though as there is no dictionary the completion character will result in a message reminding the user that word completion is not possible. Hence, none of these characters can be included in the input string. If it is necessary that one of them or one of the delimiting characters be accepted as part of the string, the programmer may define a *Quote character* which "turns off" the character which follows it. For example, if the *Quote character* is set to '@', and the remaining special characters have their default values, then the input

```
=>PS:<HALLY>GETWRD.FOR
```

would result in STRING having the value 'HALLY>GETSTR.FOR' since '<' is the default *Delete Word character* so that 'PS:' is deleted from the buffer. However,

```
=>PS:@<HALLY>GETSTR.FOR
```

results in STRING having the value 'PS:<HALLY>GETSTR.FOR'. The *Quote character* is passed via the common block / QUOTE /. If the *Quote character* is set to be one of the delimiting characters, then it is considered undefined and none of the special characters may be included in the input string. The default value for the *Quote character* is a blank, which is always one of the delimiters: i.e. the default is that the *Quote character* is undefined. If the *Quote character* is defined and it is encountered in the buffer by the subroutines GETWRD or GETNUM, it will simply be deleted since there is no purpose for it in those subroutines. Note that the restriction that none of the special characters (including the *Quote character*) can appear in a dictionary word still applies.

2.5 INPUT OF LOGICAL VARIABLES AND YES-NO ANSWERS

A logical function, GETLOG, for getting the value of a logical variable has also been provided. GETLOG is very similar to GETNUM or GETSTR but returns the logical variable LVAR via its argument list. Note that the value of the variable is not the returned value of GETLOG. rather, GETLOG, like GETNUM, GETSTR, and GETWRD returns TRUE if a value for the variable has been obtained, FALSE if the user has signalled an abort or if the buffer

has overflowed. To obtain the value of LVAR, GETLOG calls GETWRD with the simple two-word dictionary 'FALSE', 'TRUE', then sets LVAR to its appropriate value. For example, suppose the initial command 'PLOT-FLAG=' requires the user to set the value of the logical variable PLTFLG. A call to GETLOG might result in the following prompt

```
true?=>GET-FLAG=
```

to which the user must enter 'TRUE' or 'FALSE' or a response that will match one of those. Note that 'T' or 'F' is sufficient.

Another similar function, GETYN returns the answer to a 'yes or no' question. The answer is returned in a character variable, ANS, of length 1. ANS will be either 'Y' or 'N'. Like GETLOG, GETYN uses GETWRD with the simple two-word dictionary 'NO', 'YES'.

2.6 INPUT OF NUMBERS IN A SPECIFIED RANGE

Another addition to the GETWRD Package is the subroutine GNMRNG which, like GETNUM, allows the user to input a number, but also has the feature that the range of the number may be specified. If the number input, RNUM, lies outside the specified range an error message supplied by the programmer will be written. Given the lower and upper bounds for the range, RNUMLO and RNUMHI respectively, there are nine possible ways to specify the range depending whether the end-points are to be included in the range or not used at all. The range desired is specified by the integer argument IFLAG whose values for the different range specifications are:

IFLAG = 0 if neither limit is used (equivalent to GETNUM),
= 1 if lower limit used inclusively, upper limit not used: RNUMLO ≤ RNUM,
= 2 if lower limit used exclusively, upper limit not used: RNUMLO < RNUM,
= 3 if lower limit not used, upper limit used inclusively: RNUM ≤ RNUMHI,
= 4 if both limits are used inclusively: RNUMLO ≤ RNUM ≤ RNUMHI,
= 5 if lower limit is exclusive, the upper limit inclusive: RNUMLO < RNUM ≤ RNUMHI,
= 6 if lower limit not used, upper limit used exclusively: RNUM < RNUMHI,
= 7 if lower limit is inclusive, the upper limit exclusive: RNUMLO ≤ RNUM < RNUMHI,
= 8 if both limits are used exclusively: RNUMLO < RNUM < RNUMHI

2.7 SORTING THE DICTIONARY

A subroutine, SORTCT, which sorts an array of character variables into alphabetical order has been added to the GETWRD Package (see Appendix A in Reference 1 for the definition of alphabetical order). SORTCT will facilitate the implementation of programs in which the dictionary is changed during execution. If, for example, a new command is added to the dictionary, it is sufficient to add it to the end of the array DICT, then invoke SORTCT to put the new dictionary into alphabetical order. SORTCT uses a linear insertion sort modelled on an integer sort procedure by George and Liu[3].

After the dictionary is sorted or changed it is necessary to be able to identify the appropriate action to be taken when a dictionary word is identified. For example, suppose the dictionary is EXIT, GO-DOWN and GO-UP. The word GO-UP is changed to ASCEND so that the new (sorted) dictionary is ASCEND, EXIT, and GO-DOWN. But now

dictionary word 1 corresponds to what used to be dictionary word 3 so that one cannot use the position in the dictionary, DICPOS, to point the correct action to be taken since code such as

```
IF (DICPOS.EQ.1) THEN
   CALL EXIT
ELSE IF (DICPOS.EQ.2) THEN
   CALL DOWN
ELSE IF (DICPOS.EQ.3) THEN
   CALL UP
END IF
```

will result in EXIT being called when the user types ASCEND. Nor can the value of the dictionary words be used as in

```
IF (DICT(DICPOS).EQ.'EXIT') THEN
   CALL EXIT
ELSE IF (DICT(DICPOS).EQ.'GO-DOWN') THEN
   CALL DOWN
ELSE IF (DICT(DICPOS).EQ.'GO-UP') THEN
   CALL UP
END IF
```

since now none of the subroutines DOWN, EXIT or UP will be called when the user types ASCEND.

The solution is provided by an integer tag array, ITAGS returned by SORTCT. If ITAGS(J) = J for all J upon input to SORTCT, then on output ITAGS(J) will contain the position before sorting of the J^{th} dictionary word. In the example above, if on input ITAGS = (1,2,3) then on output ITAGS = (3,1,2). Suppose now that the dictionary is changed a second time: GO-DOWN will be changed to DESCEND. SORTCT is called with the unsorted dictionary ASCEND, EXIT, DESCEND and the current tag array ITAGS = (3,1,2). SORTCT returns the sorted dictionary ASCEND, DESCEND, EXIT, and the tag array ITAGS = (3,2,1). Notice that the tag array gives the *original* position of the words in the dictionary before the first sort. The appropriate code for choosing the action to be taken when a dictionary word is identified is

```
IF (ITAGS(DICPOS).EQ.1) THEN
   CALL EXIT
ELSE IF (ITAGS(DICPOS).EQ.2) THEN
   CALL DOWN
ELSE IF (ITAGS(DICPOS).EQ.3) THEN
   CALL UP
END IF
```

It is only necessary to initialize the tag array before the first dictionary sort. This may be done in SORTCT by setting the integer argument NCALL to 0. ITAGS will then be initialized so that ITAGS(J) = J, for J = 1,NWRDS, where NWRDS is the number of words in the dictionary being sorted. SORTCT will also increment NCALL by 1 so that on subsequent calls the tag array will not be initialized. Note, however, that there is no need for the elements of ITAGS to be different from one another. A convenient way to implement

synonyms (two commands causing similar action to be taken) is to set their tags to be the same. In the example above, if the command ASCEND were to be allowed as a synonym for GO-UP, then the dictionary and tags might be ASCEND, EXIT, GO-DOWN, GO-UP and ITAGS = (1,2,3,1). This example is considered in more detail in Section 3.

2.8 MODIFICATIONS TO UPCASE

The subroutine UPCASE has been modified so that its argument may now be a character string of any length: formerly its argument was only a single character. This makes UPCASE convenient for converting dictionary words to upper case if they are modified by the user, who might not type in upper case characters.

2.9 FUNCTION LEN1

LEN1 is a procedure which is called by the spelling correction subroutine CORSPL but which is likely to be convenient for more general use. LEN1 is an integer function with one argument, STRING, a character string of arbitrary length. LEN1 returns the position in STRING of the last non-blank character. It is useful when one wishes to suppress the printing of trailing blanks.

2.10 VARIABLE INITIALIZATION

Sometime during the initial execution of the procedures GETLOG, GETNUM, GETSTR, GETWRD or GETYN, the default values for the special characters, the input and output unit numbers, and the new formatting flags must be assigned to appropriate variables. The most appropriate means to do so would be a BLOCK DATA sub-program in which initial values of the variables were assigned in DATA statements. Then if changes to the variable initialization were deemed necessary by the programmer, it need only be done in one place, rather than in each of the separate procedures. Unfortunately a BLOCK DATA sub-program would cause problems if the /LIBRARY switch were used when loading the compiled GETWRD Package procedures. (On the DEC-20 computer, if the /LIBRARY switch is used, only those procedures called by the main program directly or indirectly via intermediate procedures will be loaded.) Since a BLOCK DATA sub-program is never called, it would not be loaded and the initialization would not be done. To avoid this difficulty initialization is done in the subroutine INTVAR which is, for all intents and purposes, a BLOCK DATA sub-program as it has no executable statements. However, it is called by each of GETNUM, GETSTR, and GETWRD so that it will be loaded even if the /LIBRARY switch is used. Since GETLOG and GETYN each call GETWRD, the variables will be initialized in these procedures too.

3 A SAMPLE PROGRAM

In this section a sample program illustrating many of the new features in the GETWRD Package is presented. The main program allows the user to enter one of three commands: EXIT, GO-DOWN or GO-UP. If the command is EXIT the program stops; if it is GO-DOWN the subroutine DOWN is called, if it is GO-UP the subroutine UP is called. However, before obtaining these commands the user is allowed to change the dictionary

by removing commands, defining synonyms for commands, or renaming commands. This is done in the subroutine DCCHNG which contains most of the features of interest. The arguments to DCCHNG are DICT, the dictionary to be changed; NWRDS, the current number of words in the dictionary; MXNWRD, the maximum number of words allowed in the dictionary after synonyms have been defined (the dimension of DICT is MXLWRD which sets an upper limit on the dictionary size); ITAGS the tag array which allows the main program to determine the action to be taken when a dictionary word is identified by GETWRD. Possible changes to the dictionary are obtained by entering one of the following command strings:

REMOVE *dictionary-word*

RENAME *dictionary-word new-name*

SYNONYM *dictionary-word synonym*

where *dictionary-word* is one of the words in the dictionary DICT, and *new-name* and *synonym* are character strings. REMOVE causes *dictionary-word* to be removed from the dictionary. RENAME causes *dictionary-word* to be replaced by the string *new-name*. SYNONYM causes *synonym* to be added to the dictionary as a synonym for *dictionary-word*. The command RETURN causes control to pass to the main program listed below.

```
      PROGRAM MOVE
C-----------------------------------------------------------------------C
C Sample program to illustrate new features of the GETWRD Package      C
C-----------------------------------------------------------------------C
      CHARACTER BUFFER*80, DICT(5)*7, HLPMSG*35, PROMPT*2
      INTEGER BUFPOS, DICPOS, ITAGS(5), NWRDS
      LOGICAL GETWRD

      DATA DICT/'EXIT','GO-DOWN','GO-UP',' ',' '/, NWRDS/3/,
     *  PROMPT/'=>'/, HLPMSG/'Enter one of the following commands'/,
     *  ITAGS/1,2,3,4,5/

C Call DCCHNG to allow the user to change the dictionary.
      CALL DCCHNG(NWRDS,5,DICT,ITAGS)

C Clear the buffer and get the next user command
10    BUFFER=' '
      BUFPOS=0
      IF(.NOT.GETWRD(BUFFER,NWRDS,DICT,PROMPT,HLPMSG,BUFPOS,DICPOS))
     *  GO TO 10

C If the command corresponds to the original command 'EXIT' then STOP
      IF (ITAGS(DICPOS).EQ.1) THEN
         STOP

C If the command corresponds to the original command 'GO-DOWN' then
C  call DOWN
      ELSE IF (ITAGS(DICPOS).EQ.2) THEN
         CALL DOWN

C If the command corresponds to the original command 'GO-UP' then
```

```
C  call UP
         ELSE IF (ITAGS(DICPOS).EQ.3) THEN
            CALL UP
         END IF
         GO TO 10
         END

         SUBROUTINE DCCHNG(NWRDS,MXNWRD,DICT,ITAGS)
C-------------------------------------------------------------------------C
C Subroutine which allows the user to alter a dictionary by removing      C
C  words, renaming words, or adding synonyms.                             C
C-------------------------------------------------------------------------C
         CHARACTER BUFFER*80, DICINT(4)*7, DICT(MXNWRD)*(*), PROMPT*100,
     *    HLPMSG*80, ANS
         INTEGER BUFPOS, DICPOS, DICP1, IPMT, IHLP, IQUOT, ITAGS(MXNWRD),
     *    NACPL, NACPLO, UNITIN, UNTOUT
         LOGICAL GETSTR, GETWRD, GETYN

         COMMON / FMTFLG / IPMT, IHLP, IQUOT, NACPL
         COMMON / IOUNIT / UNITIN,UNTOUT

         DATA DICINT/'REMOVE','RENAME','RETURN','SYNONYM'/, NWRD1/4/,
     *    NCALL/1/
         NACPL=MAX0(1,80/LEN(DICT(1)))

C Clear the buffer and get the command
10       BUFFER=' '
         BUFPOS=0
         PROMPT='dictionary change? =>'
         HLPMSG='Enter one of the following commands'
         IF(.NOT.GETWRD(BUFFER,NWRD1,DICINT,PROMPT(:21),HLPMSG,BUFPOS,
     *    DICPOS))GO TO 10

C If command is RETURN return to calling program
         IF (DICINT(DICPOS).EQ.'RETURN') RETURN

C Find word in DICT which is to be changed
         PROMPT='dictionary word? =>'
         HLPMSG='Which of the following words do you wish to change?'
         IF(.NOT.GETWRD(BUFFER,NWRDS,DICT,PROMPT(:19),HLPMSG,BUFPOS,
     *    DICP1))GO TO 10

C If command was REMOVE, remove the identified word from the dictionary
C  providing that there will be at least one word left.  The user is
C  asked to confirm the removal of the word.
         IF(DICINT(DICPOS).EQ.'REMOVE') THEN
            IF (NWRDS.EQ.1) THEN
               WRITE(UNTOUT,'(/2A)')' ?? YOU CANNOT REMOVE THE ONLY ',
     *          'DICTIONARY WORD'
               GO TO 10
            END IF
            BUFFER=' '
            BUFPOS=0
            L1=LEN1(DICT(DICP1))
```

```
            PROMPT='(1X,"Do you really want to delete the command '//
     *       DICT(DICP1)(:L1)//'?"/1X,"Answer Y or N:  =>"$)'
            IPMT=1
            IHLP=1
            IQUOT=1
            NACPLO=NACPL
            NACPL=0
            IF(.NOT.GETYN(BUFFER,PROMPT,HLPMSG,BUFPOS,ANS,IER))GO TO 10
            IF (ANS.EQ.'Y') THEN
               DO 20 I=1,LEN(DICT(1))
                  DICT(DICP1)(I:I)='~'
20             CONTINUE
               CALL SORTCT(NWRDS,DICT,ITAGS,BUFFER,NCALL,IER)
               NWRDS=NWRDS-1
            END IF
            IPMT=0
            IHLP=0
            IQUOT=0
            NACPL=NACPLO

C If the command is RENAME, get the new word from the user.
         ELSE IF(DICINT(DICPOS).EQ.'RENAME') THEN
            PROMPT='new word name? =>'
            HLPMSG='Enter the new word name to replace '//
     *       DICT(DICP1)
            IF(.NOT.GETSTR(BUFFER,PROMPT(:17),HLPMSG,BUFPOS,DICT(DICP1),
     *       LENSTR,IER))GO TO 10
            CALL UPCASE(DICT(DICP1))
            CALL SORTCT(NWRDS,DICT,ITAGS,BUFFER,NCALL,IER)

C If command is SYNONYM, add the synonym to the dictionary and set ITAGS
C  so that the new word is equivalent to the old word.
         ELSE IF(DICINT(DICPOS).EQ.'SYNONYM') THEN
            IF (NWRDS.EQ.MXNWRD) THEN
               WRITE(UNTOUT,'(/2A)')' ?? No more room in the ',
     *          'dictionary for synonyms'
               GO TO 10
            END IF
            PROMPT='synonym name? =>'
            HLPMSG='Enter the synonym for the word '//DICT(DICP1)
            NWRDS=NWRDS+1
            IF(.NOT.GETSTR(BUFFER,PROMPT(:16),HLPMSG,BUFPOS,DICT(NWRDS),
     *       LENSTR,IER))GO TO 10
            CALL UPCASE(DICT(NWRDS))
            ITAGS(NWRDS)=ITAGS(DICP1)
            CALL SORTCT(NWRDS,DICT,ITAGS,BUFFER,NCALL,IER)
         END IF

C Call DICCHK to check the sorted dictionary
         CALL DICCHK(NWRDS,DICT)
         GO TO 10
         END
```

Comments:

1) The dictionary DICT initially has only three words but is dimensioned to five so that two user-defined synonyms may be added. Thus, in DCCHNG MXLWRD is five and NWRDS is initially three.

2) A user abort at any level within DCCHNG causes the buffer to be cleared and a new dictionary-modification command to be prompted.

3) Upon entering DCCHNG, NACPL is set so that several available commands can be written on one line in response to the *Help character*. However, in subroutine GETYN all necessary information is contained in the prompt. It is desired that the response to a *Help character* is just the reprinting of the prompt. This is achieved by passing an empty help message, setting IHLP to one, and suppressing the printing of available commands by setting NACPL to zero. This causes the *Help character* to be completely ignored since a WRITE statement with an empty format string does nothing at all. If IHLP were not set to one, the blank help message would be printed causing an extra blank line on the terminal screen. After returning from GETYN, NACPL is reset to its previous value and IHLP is reset to zero.

4) When setting the value for PROMPT before the call to GETYN, the integer function LEN1 is called to determine the last non-blank character in DICT(DICP1). It is then possible to avoid several superfluous blanks between the dictionary word and the question mark: e.g.

   Do you really want to delete the command GO-UP?

   rather than

   Do you really want to delete the command GO-UP ?

5) The prompt used when asking for confirmation of the removal of a word (passed to GETYN) covers two lines so that it must be passed as a format specifier. Hence, before entering GETYN, IPMT and IQUOT are set to one. They are reset to zero after control is returned from GETYN.

6) The contents of a word to be removed are set to '~~~~~~~' (whatever length is necessary). Since ~ is the last character in the allowed alphabet of characters, this ensures that the word will be sorted to the end of the dictionary where it will be ignored when NWRDS is set to NWRDS-1.

7) ITAGS is initialized in the main program and SORTCT is always called with NCALL > 0. If NCALL is initialized to zero so that ITAGS is initialized during the first call to SORTCT, a problem arises if the first user command is SYNONYM. Suppose the first user command is 'SYNONYM GO-DOWN ASCEND'. In that case, the first time SORTCT is called it is passed EXIT, GO-DOWN, GO-UP, ASCEND and NWRDS = 4. ITAGS is initialized to ITAGS = (1,2,3,4,*) with the star standing for an indeterminate fifth element. This overwrites the value ITAGS(4) = 1 which was set just prior to the call to SORTCT. In general, it is always safest to initialize the tag array in the procedure in which the dictionary commands are used (here the main program MOVE) rather than in the subroutine SORTCT.

8) The strings *new-name* and *synonym* are obtained using GETSTR. Since these strings will become dictionary words and none of the special characters are allowed in dictionary words, there is no point in defining a *Quote character* to allow inclusion of special characters in these strings. Hence, the *Quote character* is left undefined.

The following is a sample run of the program MOVE on the DEC 20 computer at DREA. The subroutines UP and DOWN, which are not shown here, cause the simple messages 'MOVING UP...' and 'MOVING DOWN...' to be written on the terminal screen.

```
@EXE PS:<HALLY>MOVE,PS:<HALLY>GETWRD/LIB
LINK:   Loading
[LNKXCT MOVE execution]

dictionary change? =>?

Enter one of the following commands
REMOVE    RENAME    RETURN    SYNONYM

dictionary change? =>syn ?

Which of the following words do you wish to change?
EXIT      GO-DOWN   GO-UP

dictionary word? =>SYN go-up ?

Enter the synonym for the word GO-UP

synonym name? =>SYN GO-UP ascend

dictionary change? =>ren ?

Which of the following words do you wish to change?
ASCEND    EXIT      GO-DOWN   GO-UP

dictionary word? =>REN go-d ?

Enter the new word name to replace GO-DOWN

new word name? =>REN GO-D descend

dictionary change? =>rem ?

Which of the following words do you wish to change?
ASCEND    DESCEND   EXIT      GO-UP

dictionary word? =>REM go-up

Do you really want to delete the command GO-UP?
Answer Y or N:  =>?


Do you really want to delete the command GO-UP?
Answer Y or N:  =>y
```

```
dictionary change? =>return

=>?

Enter one of the following commands
ASCEND    DESCEND   EXIT

=>ascend

MOVING UP...

=>descend

MOVING DOWN...

=>exit
CPU time 1.05   Elapsed time 1:27.19
●
```

4 CONCLUDING REMARKS

The implementation and use of several new features in the GETWRD Package of FORTRAN 77 procedures have been described. The new features not only enhance the user friendliness of the package, but provide greater flexibility and ease of use to the programmer. This makes the GETWRD Package an even more useful tool for FORTRAN programmers wishing to implement command languages.

Acknowledgement

The author would like to thank Dr.D.D.Ellis for his helpful remarks and criticisms and for suggesting and supplying the original code (now somewhat modified) for the subroutine SORTCT.

Appendix A

Implementation of the New Features

In this Appendix the implementation of the new GETWRD Package features is discussed in detail.

A.1 CHANGES IN DATA STRUCTURES

To maintain compatibility with existing programs which make use of the GETWRD Package none of the data structures described in the GETWRD Package Manual have been changed. However, the following new variables have been defined and are used in several of the procedures.

IHLP = An integer flag used to indicate whether the character variable HLPMSG is to be interpreted as a simple character string or as a format specifier (see Section 2.1).
= 0, if HLPMSG is interpreted as a simple character string
= 1, if HLPMSG is interpreted as a format specifier.

IHLP is initialized to 0 in a data statement in the subroutine INTVAR. It is passed to other procedures via the common block / FMTFLG /.

IPMT = An integer flag used to indicate whether the character variable PROMPT is to be interpreted as a simple character string or as a format specifier (see Section 2.1).
= 0, if PROMPT is interpreted as a simple character string
= 1, if PROMPT is interpreted as a format specifier.

IPMT is initialized to 0 in a data statement in the subroutine INTVAR. It is passed to other procedures via the common block / FMTFLG /.

IQUOT = Integer flag used to indicate whether double quotes in the variables PROMPT and HLPMSG are to be replaced by single quotes.
= 0, if no substitution is to be made
= 1, if double quotes are replaced by single quotes.

IPMT is initialized to 0 in a data statement in the subroutine INTVAR. It is passed to other procedures via the common block / FMTFLG /.

NACPL = Integer whose value indicates the number of allowed dictionary words to be written on each line following a user plea for help via the *Help character* (see Section 2.1). If NACPL is zero none of the allowed dictionary words will be written. NACPL is initialized to 1 in a data statement in the subroutine INTVAR. It is passed to other procedures via the common block / FMTFLG /.

QOTCHR = Character variable of length 1 containing the character to be used as the special *Quote character* (see Section 2.4). If QOTCHR is the same as one of the delimiters then it is considered to be turned off and will have no effect.

QOTCHR is initialized to ' ' (always a delimiter) in a data statement in the subroutine INTVAR. It is passed to other procedures via the common block / QUOTE /.

A.2 MODIFICATIONS TO INDIVIDUAL GETWRD PACKAGE PROCEDURES

In this section the modifications to each of the existing GETWRD Package procedures are described. The procedures CLBUFF, DLIMIT, INSBUF, MDICW, and NXTWRD have had no modifications.

A.2.1 Subroutine DELETE

The subroutine DELETE is called by the subroutine PRMPT after user input. It checks the buffer for the special *Delete Letter* and *Delete Word characters* and then modifies the buffer accordingly. DELETE has been modified to allow the special *Quote character* to "turn off" the *Delete Letter* and *Delete Word characters*.

New common block: COMMON / QUOTE / QOTCHR

Internal variables:

DELON = Logical variable which is TRUE if the current delete character is not quoted.

DELPOS = Integer whose value is the position in BUFFER(IBUFF:LEN(IBUFF)) of the next *Delete Letter* or *Delete Word character*.

IBUFF = Integer whose value is the position of the last *Delete Letter* or *Delete Word character* found in the buffer. Initially IBUFF is BEGWRD+1.

IBPDM2 = IBUFF+DELPOS-2 = Position in the buffer of the character preceding the *Delete Letter* or *Delete Word character*.

QUOTON = Logical variable which is FALSE if QOTCHR is a delimiter, TRUE if it is not. QUOTON is use to avoid repeated calls to DLIMIT.

Algorithm:

```
Begin
  Determine value of QUOTON
  Set IBUFF to BEGWRD+1
  While there is another Delete Letter character beyond IBUFF in BUFFER do
    DELPOS = position in BUFFER(IBUFF:LEN(BUFFER)) of Delete Letter character
    If the Delete Letter character is the first character in the word delete
     the Delete Letter character only
    Else If QUOTON is TRUE and if the Delete Letter character is quoted set
     IBUFF to IBUFF+DELPOS
    Else
      Delete Letter character and the preceding character
      If the preceding character was quoted, also delete the Quote character
      If deleted characters before BUFPOS, adjust BUFPOS
```

```
    End If
  End do

  Set IBUFF to BUFPOS+1
  While there is another Delete Letter character beyond IBUFF in BUFFER do
    DELPOS = position in BUFFER(IBUFF:LEN(BUFFER)) of Delete Word character
    If the Delete Word character is the first character in BUFFER then delete
     the Delete Word character
    Else if QUOTON is TRUE and the Delete Letter character is quoted set IBUFF
     to IBUFF+DELPOS
    Else
      Search for backwards in the buffer for the first non-quoted Delimiting
       character
      Delete all characters from the Delete Letter character back to but not
       including the Delimiting character
      If deleted characters before BUFPOS, adjust BUFPOS
    End If
  End do
  Return
End
```

A.2.2 Logical Function DICCHK

DICCHK is used to check the dictionary for errors. The only change that has been made is to check that the *Quote character* does not appear in any dictionary words. This is implemented simply by extending the length of the string ENDC1 to 6 and including the QOTCHR in it (See the GETWRD Package Manual, Appendix D.3). There is no need to check to see if the *Quote character* is turned on, since if it is turned off it is equal to a delimiter which also is not allowed in a dictionary word. The common block / QUOTE / is also included to pass the *Quote character* to DICCHK.

A.2.3 Logical Function GETNUM

The logical function GETNUM used to interpret the next word of input in the buffer as a number, has been modified in the following ways.

1) The common blocks / FMTFLG / and / QUOTE / have been included to pass the formatting flags IHLP, IPMT, IQUOT, and NACPL and the *Quote character*.

2) A call to subroutine INTVAR is included to initialize variables (see Section 2.10).

3) If IQUOT is 1, CHQOTE is called to change double quotes in HLPMSG and PROMPT to single quotes.

4) If a *Quote character* is found in the word of input it is removed since it has no special meaning in this context.

5) If a *Help character* is found, the help message is written according to the value of IHLP (see Section 2.1). If IHLP is 0, HLPMSG is written as a string:

```
WRITE(UNITNO,'(/1X,A)')HLPMSG
```

If IHLP is 1, HLPMSG is used as a format specifier:

```
WRITE(UNITNO,HLPMSG)
```

A.2.4 Logical Function GETWRD

The logical function GETWRD used to match the next word of input in the buffer with one of the words in a dictionary, has been modified in the following ways:

1) The common blocks / FMTFLG / and / QUOTE / have been included to pass the formatting flags IHLP, IPMT, IQUOT, and NACPL and the *Quote character*.

2) A call to subroutine INTVAR is included to initialize variables (see Section 2.10).

3) If IQUOT is 1, CHQOTE is called to change all double quotes in HLPMSG and PROMPT to single quotes.

4) If a *Quote character* is found in the word of input it is removed since it has no special meaning in this context.

5) If no word in the dictionary matches the word input, logical function CORSPL is called to determine whether a dictionary word could be matched if allowance was made for simple typographical errors (see Section 2.2). If a match is found the user is asked whether the the corrected word should be matched with the dictionary word.

Algorithm:

```
Begin
  Call INTVAR to initialize variables
  If IQUOT is 1 changed double quotes to single quotes in HLPMSG and PROMPT
  If no characters past present point in BUFFER then
    Call PRMPT to get user input
    If PRMPT returns FALSE return GETWRD = FALSE
  End if

  Repeat
    Get next character in BUFFER
    Change character to upper case
    If character is word terminator
      If valid word has not been found then
        Clear BUFFER from present position to end
        Call PRMPT to prompt user and read input
      End if
    Else if character is Abort character set IER to -1 and return
    Else if character is Quote character remove it
    Else
      Find first possible matching word in the dictionary
      If no valid word can be found then
```

```
            Call CORSPL to check for typographical errors
            If none then
              Clear BUFFER from present position to end
              Call PRMPT to prompt user and read input
              If PRMPT returns FALSE return GETWRD = FALSE
            End if
          End if
        End if
      Until finished word
      GETWRD = TRUE
      Return position of word in dictionary
    End
```

A.2.5 Logical Function PRMPT

The logical function PRMPT is used to prompt the user and then to read new input from the terminal. Subroutine DELETE is then called to process any *Delete Letter* or *Delete Word characters* in the buffer. PRMPT has been updated to allow the character variable PROMPT to be interpreted either as a simple character string or as a format specifier (see Section 2.1). If integer flag IPMT (passed via common block / FMTFLG /) is 0, PROMPT is interpreted as a string, while if IPMT is 1, PROMPT is interpreted as a format specifier. The WRITE statements in PROMPT are as follows.

1) If IPMT = 0 and BUFPOS = 0, then PROMPT is written as a string and the contents of the buffer are not written.

```
WRITE(UNTOUT,'(/1X,A$)')PROMPT
```

2) If IPMT = 0 and BUFPOS > 0, then PROMPT is written as a string followed by the contents of the buffer up to BUFPOS:

```
WRITE(UNTOUT,'(/1X,2A$)')PROMPT,BUFFER(:BUFPOS)
```

3) If IPMT = 1 and BUFPOS = 0, then PROMPT is used as a format specifier and the contents of the buffer are not written:

```
WRITE(UNTOUT,PROMPT)
```

4) If IPMT = 1 and BUFPOS > 0, then PROMPT is used as a format specifier and the contents of the buffer up to BUFPOS are then written in a separate WRITE statement:

```
WRITE(UNTOUT,PROMPT)
WRITE(UNTOUT,'(''+''A$)')BUFFER(:BUFPOS)
```

A.2.6 Subroutine UPCASE

The subroutine UPCASE is used to convert characters from lower case to upper case before attempting to match them with characters in a dictionary word. However, if

the dictionary is to be modified by the user it is convenient to use UPCASE to ensure that new dictionary words are in upper case. Hence, UPCASE has been modified to accept strings of arbitrary length and convert them to upper case. As pointed out in the GETWRD Package Manual, Appendix E, note that UPCASE is intrinsically non-ANSI since lower case characters are not included in the ANSI standard FORTRAN character set.

A.2.7 Logical Function WRDTRM

The logical function WRDTRM is used to process the special characters, *Help character*, *Completion character* and any *Delimiting characters*. WRDTRM has been modified to allow the character variable HLPMSG to be interpreted either as a simple character string or as a format specifier (see Section 2.1). If integer flag IHLP (passed via common block / FMTFLG /) is 0, HLPMSG is interpreted as a string, while if IHLP is 1 HLPMSG is interpreted as a format specifier. In addition, WRDTRM has been changed so that in response to the *Help character*, the number of allowed dictionary words written per line is specified by the variable NACPL (also passed via the common block / FMTFLG /). The separation between each column of allowed dictionary words is three spaces. It is up to the programmer to ensure that the horizontal extent of the commands will not overflow the terminal screen.

The WRITE statements for the help message in WRDTRM are now as follows:

1) If IHLP is 0, HLPMSG is written as a string:

```
WRITE(UNITNO,'(/1X,A)')HLPMSG
```

2) If IHLP is 1, HLPMSG is used as a format specifier:

```
WRITE(UNITNO,HLPMSG)
```

WRDTRM has also been modified so that when the completion character is used but fails to match a unique word in the dictionary, WORDOK is called to attempt to correct the word. If WORDOK is successful, the remainder of the buffer is processed normally. This prevents the loss of additional input in the buffer. If WORDOK fails the user is prompted for new input and the remainder of the buffer is lost. Previously an error message was written immediately and the user prompted for new input; any words in the buffer beyond the mistake were lost.

A.3 NEW PROCEDURES IN THE GETWRD PACKAGE

A.3.1 Logical Function CHQOTE

Purpose

CHQOTE changes all double quotes in its argument string to single quotes. It is used to pre-process help messages, error messages, and prompts when these are format specifiers (see Section 2.1).

Arguments: STRING

where

STRING = Character variable containing the string whose double quotes are to be changed.

Internal variables:

LQOTE = Integer variable whose value is the position in STRING of a double quote.

Algorithm:

```
Begin
  Do while STRING contains a double quote
    Change first double quote to single quote
  End do
End
```

A.3.2 Logical Function CORSPL

Purpose

CORSPL attempts to detect a user typographical error if GETWRD cannot find a match between the word in the buffer and any of the dictionary words. CORSPL returns TRUE if there is a word in the dictionary which would match a dictionary word if one of the following modifications were made:

1) two letters were transposed,
2) one letter was changed to another letter,
3) one letter was added, or
4) one letter was deleted,

and if the user has confirmed that the corrected word is, indeed, the word that was meant. The algorithm is derived from one due to Durham, Lamb and Saxe[2].

Arguments: NWRDS, DICT, BUFFER, BEGWRD, BUFPOS, DICPOS

where

NWRDS, DICT, and BUFFER are as in GETWRD Package Manual, Appendix C

BEGWRD = Integer variable containing the position in BUFFER of the first character in the word being checked.

BUFPOS = Integer variable containing the current position in BUFFER (as in GETWRD, GETNUM, etc.). When CORSPL is called BUFPOS will NOT be the position of the first delimiter before the current word as it is in most other procedures; rather, it will be the position in BUFFER of the first character in the current word which caused a mismatch with the dictionary words.

DICPOS = The position in the dictionary of the word matched if CORSPL was successful. If CORSPL returns FALSE, then DICPOS will have the value it had on input.

Internal variables:

ENDWRD = Integer variable whose value is the position in BUFFER of the last character in the current word of input.

LCORW = Integer variable containing the length of the corrected word of user input if a unique match is found.

LDICT = Integer variable containing the length of the current dictionary word not including trailing blanks.

LMATCH = The number of letters for which the word of user and the current dictionary word match. Note that LMATCH < LWRD since if LMATCH = LWRD there would have been a match in GETWRD (or ambiguous words) and CORSPL would not have been called.

LWRD = Integer variable containing the length of the word of user input.

NBUFF = Integer variable containing the length of BUFFER

Algorithm:

```
Begin
  Save DICPOS in DPSAV
  CORSPL = TRUE
  Find the last character in the word of user input
  Convert remaining characters of user input to upper case
  Calculate LWRD
  Do for each word in the dictionary
    Set DICPOS to the position in DICT of the current word
    Calculate LDICT, the length of the word minus trailing blanks
    If LWRD > LDICT+1, go to next iteration of loop since no match can be made
    Find LMATCH
    If LWRD = LMATCH+1 (there is a match by the "additional letter" or the
     "incorrect letter" criteria) then
      If user confirms the match correct BUFFER and return
      Else go to the next iteration of the loop
      End if
    End if
    If LMATCH+2 ≤ LDICT and LWRD ≤ LDICT then
      Check for transposition of letters.
      If match is found
        If user confirms the match correct BUFFER and return
        Else go to the next iteration of the loop
        End if
      End if
    End if
    If LWRD ≤ LDICT then
      Check for an incorrect letter in the word
```

```
        If match is found
          If user confirms the match correct BUFFER and return
          Else go to the next iteration of the loop
          End if
        End if
      End if
      If LWRD-1 ≤ LDICT then
        Check for an additional letter in the word
        If match is found
          If user confirms the match correct BUFFER and return
          Else go to the next iteration of the loop
          End if
        End if
      End if
      If LWRD+1 ≤ LDICT then
        Check for a missing letter in the word
        If match is found
          If user confirms the match correct BUFFER and return
        End if
      End if
    End do
    Reset DICPOS to DPSAV
    Return CORSPL = FALSE
  End
```

A.3.3 Logical Function GETLOG

Purpose

GETLOG interprets the next word of user input as a logical variable, returning TRUE if successful and FALSE if the user has signalled an abort (by entering the *Abort character*) or if the buffer has overflowed. GETLOG calls GETWRD to match the word of input with the two word dictionary 'FALSE','TRUE'.

Arguments: BUFFER, PROMPT, HLPMSG, BUFPOS, LVAR, IER

where

BUFFER, PROMPT, HLPMSG, BUFPOS are as in GETWRD Package Manual, Appendix C

LVAR = Logical variable containing the value corresponding to the word of input. The value of LVAR will be changed from its input value only if GETLOG is successful: i.e. if GETLOG returns FALSE, LVAR will return the same value it had on input.

IER = Integer error flag
= -1 if there is a controlled abort
= -2 if the buffer has overflowed

Algorithm:

```
Begin
```

```
    Set IER to 0
    Call GETWRD to match the next word in BUFFER with FALSE or TRUE
    If GETWRD returns TRUE then
      GETLOG = TRUE
      If word matched was TRUE, LVAR = TRUE
      Else LVAR = FALSE
      End if
      IER = 0
    Else
      GETLOG = FALSE
      IER = DICPOS
    End if
    Return
  End
```

A.3.4 Logical Function GETSTR

Purpose

GETSTR interprets the next word of user input as a simple character string, returning TRUE if successful and FALSE if the user has signalled an abort (by entering the *Abort character*) or if the buffer has overflowed. If a *Help character* is found, the help message is written according to the value of IHLP (see Section 2.1). If IHLP is 0, HLPMSG is written as a string:

```
WRITE(UNITNO,'(/1X,A)')HLPMSG
```

If IHLP is 1, HLPMSG is used as a format specifier:

```
WRITE(UNITNO,HLPMSG)
```

If a *Completion character* is found an error message is written followed by the help message as word completion is not possible in GETSTR.

Arguments: BUFFER, PROMPT, HLPMSG, BUFPOS, STRING, LENSTR, IER

where

BUFFER, PROMPT, HLPMSG, BUFPOS are as in GETWRD Package Manual, Appendix C

STRING = Character variable in which the input string is returned. The value of STRING will be changed from its input value only if GETSTR is successful: i.e. if GETSTR returns FALSE, STRING will return the same value it had on input.

LENSTR = Integer variable containing the length of the input string not counting trailing delimiting characters.

IER = Integer error flag
= -1 if there is a controlled abort
= -2 if the buffer has overflowed

Common blocks: (see the GETWRD Package Manual, Section 5)

```
COMMON / ENDCHR / HLPCHR,CMPLT,DELLTR,DELWRD,ABORT
COMMON / FMTFLG / IPMT,IHLP,IQUOT,NACPL
COMMON / QUOTE / QOTCHR
COMMON / IOUNIT / UNITIN,UNTOUT
```

Internal variables:

QUOTON = Logical variable which is FALSE if QOTCHR is a delimiter, TRUE if it is not. QUOTON is use to avoid repeated calls to DLIMIT.

Algorithm:

```
Begin
  Call INTVAR to initialize variables
  Determine value of QUOTON
  If IQUOT = 1 change double quotes in PROMPT and HLPMSG to single quotes
  Initialize BEGWRD
  If no characters past the current point BUFFER then
    Call PRMPT to get user input
    If PRMPT returns FALSE return GETSTR = FALSE
  End if

  Repeat
    Get the next character
    If the character is a Quote character, set BUFPOS to BUFPOS+1
    Else if the character is a delimiting character then
      Set value of LENSTR
      Find true length of input string not counting Quote characters
      Check that input string will fit in the variable STRING
      If it will not then
        Write an error message
        Call PRMPT to prompt the user for input (the user must use the
         Delete Letter or Delete Word characters to shorten the string)
        If PRMPT returns FALSE, return GETSTR = FALSE
      Else
        Strip the QOTCHRs from the word in BUFFER
        GETSTR = TRUE
        Return
      End if
    Else if current character is the Help character then
      Write HLPMSG according to the value of IHLP
      Call PRMPT to get user input
      If PRMPT returns FALSE, return GETSTR = FALSE
    Else if current character is the Completion character then
      Write an error message
      Write HLPMSG according to the value of IHLP
      Call PRMPT to get user input
      If PRMPT returns FALSE, return GETSTR = FALSE
    Else if current character is the Abort character then
      GETSTR = FALSE and IER = -1
      Return
```

```
    End if
  Until end of BUFFER word
  Return
End
```

A.3.5 Logical Function GETYN

Purpose

GETYN interprets the next word of user input as one of YES or NO, returning TRUE if successful and FALSE if the user has signalled an abort (by entering the *Abort character*) or if the buffer has overflowed. GETYN calls GETWRD to match the word of input with the two word dictionary 'NO','YES'.

Arguments: BUFFER, PROMPT, HLPMSG, BUFPOS, ANS, IER

where

BUFFER, PROMPT, HLPMSG, BUFPOS are as in GETWRD Package Manual, Appendix C

ANS = Character variable of length 1 containing 'Y' if the word of input matched 'YES' and 'N' if it matched 'NO'.

IER = Integer error flag
= -1 if there is a controlled abort
= -2 if the buffer has overflowed

Algorithm:

```
Begin
  Set IER to 0
  Call GETWRD to match the next word in BUFFER with NO or YES
  If GETWRD returns TRUE then
    GETYN = TRUE
    ANS = first letter of dictionary word matched
    IER = 0
  Else
    GETYN = FALSE
    IER = DICPOS
  End if
  Return
End
```

A.3.6 Logical Function GNMRNG

Purpose

GNMRNG interprets the next word of user input as a number but only accepts it if it is within a range specified by the arguments RNUMLO and RNUMHI. GNMRNG returns TRUE if successful and FALSE if the user has signalled an abort (by entering the *Abort character*)

or if the buffer has overflowed. GNMRNG uses GETNUM to obtain the number from the buffer.

Arguments: RNUMLO, RNUMHI, BUFFER, PROMPT, HLPMSG, ERRMSG, BUFPOS, RNUM, IFLAG

where

BUFFER, PROMPT, HLPMSG, BUFPOS are as in GETWRD Package Manual, Appendix C

RNUMLO = Real variable whose value is the lower end of the allowed range for the input number.

RNUMHI = Real variable whose value is the upper end of the allowed range for the input number.

ERRMSG = Character variable used to write an error message if the input number is out of range. ERRMSG is treated just as HLPMSG: it is interpreted as a simple string if IHLP is 0, as a format specifier if IHLP is 1.

RNUM = Real variable containing the number input by the user. The value of RNUM will be changed from its value before GNMRNG was called only if GNMRNG is successful: i.e. if GNMRNG returns FALSE, RNUM will return the same value it had before GNMRNG was called.

IFLAG = Integer flag used on input to indicate how the end-points of the range are to be treated, and on output to signal a user abort or buffer overflow. On input:
- = 0, if neither limit is used (equivalent to GETNUM)
- = 1, if lower limit used inclusively, upper limit not used: RNUMLO ≤ RNUM
- = 2, if lower limit used exclusively, upper limit not used: RNUMLO < RNUM
- = 3, if lower limit not used, upper limit used inclusively: RNUM ≤ RNUMHI
- = 4, if both limits are used inclusively: RNUMLO ≤ RNUM ≤ RNUMHI
- = 5, if lower limit is exclusive, the upper limit inclusive: RNUMLO < RNUM ≤ RNUMHI
- = 6, if lower limit not used, upper limit used exclusively RNUM < RNUMHI
- = 7, if lower limit is inclusive, the upper limit exclusive RNUMLO ≤ RNUM < RNUMHI
- = 8, if both limits are used exclusively: RNUMLO < RNUM < RNUMHI

On output:
- = -1 if there is a controlled abort
- = -2 if the buffer has overflowed
- = -3 if RNUMLO > RNUMHI and IFLAG on input was 4, 5, 7, or 8

Common blocks: (see the GETWRD Package Manual, Section 5)

```
COMMON / FMTFLG / IPMT,IHLP,IQUOT,NACPL
COMMON / IOUNIT / UNITIN,UNTOUT
```

Internal variables:

BPSAV = Integer variable used to save the input value of the buffer position BUFPOS.

INRNG = Logical variable which is TRUE if the input number is in range.

RNMSAV = Real variable used to save the input value of the variable RNUM so that it may be restored if the user aborts a call to GETNUM.

Algorithm:

```
Begin
  If IQUOT is 1 replace double quotes in ERRMSG by single quotes
  If RNUMLO > RNUMHI and IFLAG is 4,5,7 or 8 set IER to -3 and return
  Save RNUM in RNMSAV and BUFPOS in BPSAV
  Repeat
    Call GETNUM to interpret the next word in BUFFER as a number
    If GETNUM returns FALSE, return GNMRNG = FALSE
    If input number is in range return GNMRNG = TRUE
    Else write ERRMSG according to value of IHLP
    End If
    Reset RNUM to RNMSAV and BUFPOS to BPSAV
    Clear BUFFER past BPSAV
  Until number in range is found
End
```

A.3.7 Subroutine INTVAR

Purpose

INTVAR is used to initialize the default values for the special characters, for the formatting flags, and for the logical input and output devices. See Section 2.10.

Arguments: none

Common blocks: (see Appendix A.1 and the GETWRD Package Manual, Section 5)

```
COMMON / FMTFLG / IPMT,IHLP,IQUOT,NACPL
COMMON / IOUNIT / UNITIN,UNTOUT
```

Algorithm:

INTVAR contains no executable statements. It is used in preference to a BLOCK DATA sub-program to initialize variables using DATA statements.

A.3.8 Integer Function LEN1

Purpose

LEN1 returns the length of a character variable not including trailing blanks. Note that on many machines a character variable which is not initialized will be filled with characters other than spaces so that in this case LEN1 will not return 0.

Arguments: STRING

where

STRING = Character variable whose length is to be determined.

Algorithm:

```
Begin
  Do for each character in string starting from the end
    If character is not a blank, return LEN1 = position in STRING
  End do
  Return LEN1 = 0
End
```

A.3.9 Logical Function QUOTED

Purpose

QUOTED determines whether a character in the buffer is quoted or not. It assumes that the *Quote character* is turned on. It is not sufficient to check whether the preceding character is a quote character since the quote character itself may be quoted. QUOTED returns TRUE if there are an odd number of *Quote characters* immediately preceding the character to be checked.

Arguments: BUFFER, BUFPOS

where

BUFFER as in GETWRD Package Manual, Appendix C

BUFPOS = Integer variable whose value is the position in BUFFER of the character to be checked.

Common block: COMMON / QUOTE / QOTCHR (see Section A.1)

Algorithm:

```
Begin
  Find number of Quote characters preceding BUFPOS in BUFFER
  If even number of Quote characters QUOTED = FALSE
  Else QUOTED = TRUE
  End If
  Return
End
```

A.3.10 Subroutine SORTCT

Purpose

SORTCT is a tagged linear insertion sort procedure designed to sort an array of character variables into alphabetic order. Alphabetic order is defined by the ASCII collating sequence (see the GETWRD Package Manual, Appendix A). SORTCT is a modification of an algorithm by George and Liu[3].

Arguments: N, CHARS, ITAGS, CTEMP, NCALL, IER

where

N = Integer variable containing the length of the character array to be sorted.

CHARS = Character array of length N containing the variables to be sorted.

ITAGS = Integer array of length N containing the tags by which one can determine the original position of any word in CHARS (see Section 2.7).

CTEMP = Character variable whose length is at least as long as the elements of CHARS. CTEMP is used as a temporary variable when swapping elements of CHARS. For most applications using the GETWRD Package the buffer variable BUFFER is a convenient variable to use for CTEMP.

NCALL = Integer variable used as a flag. If NCALL = 0, the tag array ITAGS will be initialized to the identity permutation: ITAGS(J) = J, J = 1,N. Otherwise ITAGS the input value of ITAGS will be used (see comments in Section 2.7).

IFLAG = Integer variable use as an error flag:
= 0, if no error occurred
= 1, if N ≤ 0
= 2, if CTEMP is not long enough.

Internal variables:

ITEMP = Integer variable used as temporary variable when swapping elements of ITAGS.

Algorithm:

```
Begin
  If N ≤ 0 return IER = 1
  If LEN(CTEMP) < LEN(CHARS(1)) return IER = 2
  Else IER = 0
  End If

  If NCALL = 0, initialize ITAGS to the permutation identity

  Do for each element of CHARS (and ITAGS) from 2 to N (K = 2,N)
```

```
        Insert the current element of CHARS into its correct position in
         CHARS(1)-CHARS(K-1). (Note that this sub-array is already
         sorted).
        For every pair of elements of CHARS swapped, swap the same pair of
         elements in ITAGS
      End do
      Return
    End
```

A.3.11 Logical Function WORDOK

Purpose

WORDOK asks the user for confirmation that a dictionary word matched by CORSPL with the word of user input is, indeed, the word which the user wanted. If so makes the correction in BUFFER and returns TRUE. Else returns FALSE. If the user types an *Abort character* when prompted, WORDOK returns TRUE but DICPOS is -1. This will cause CORSPL to return TRUE to GETWRD, which then interprets the value of DICPOS as an abort flag.

Arguments: BUFFER, BUFPOS, BEGWRD, ENDWRD, DICWRD, LDICT, LCORW, DICPOS

where

BUFFER, BUFPOS, DICPOS are as in GETWRD Package Manual, Appendix C

BEGWRD = Integer variable containing the position in BUFFER of the first character in the word being checked.

DICWRD = Character variable containing the word in the dictionary to be matched.

LDICT = Integer variable containing the length of DICWRD not including trailing blanks.

LCORW = Integer variable containing the length of the corrected word of user input: i.e. DICWRD(:LCORW) is what the user should have typed.

Common blocks: (see the GETWRD Package Manual, Section 5)

```
COMMON / ENDCHR / HLPCHR,CMPLT,DELLTR,DELWRD,ABORT
COMMON / IOUNIT / UNITIN,UNTOUT
```

Internal variables:

ANS = Character variable of length 3 used to obtain a Yes-No answer from the user.

Algorithm:

```
Begin
  Write prompt asking for confirmation of the correction of the user's input
  Repeat
    Obtain answer
```

```
        If answer is YES
          WORDOK = TRUE
          Correct BUFFER setting BUFPOS to the delimiter following the word
        Else if answer is NO
          WORDOK = FALSE
        Else if user has signalled an abort
          WORDOK = TRUE
          DICPOS = -1
        Else Write error message
        End if
      Until valid answer received
      Return
    End
```

References

1. Hally,D.; Dent,C.A.; GETWRD Package Manual, DREA TM/84/D

2. Durham,I.; Lamb,D.A.; Saxe,J.B.; "Spelling Correction in User Interfaces", Comm. A.C.M. 26, 764(1983)

3. George,A.; Liu,J.W.; Computer Solutions of Large Sparse Positive Definite Systems, Ch.6, Prentice-Hall Inc., Englewood Cliffs, N.J., 1981

Subject Index

ADA 142698

UNLIMITED DISTRIBUTION

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R & D
(Security classification of title, body of abstract and indexing annotation must be entered when the overall document is classified)

1. ORIGINATING ACTIVITY Defence Research Establishment Atlantic	2a. DOCUMENT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP

3. DOCUMENT TITLE

GETWRD PACKAGE UPDATE: NEW FEATURES AND MODIFICATIONS TO THE GETWRD PACKAGE

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)
TECHNICAL COMMUNICATION

5. AUTHOR(S) (Last name, first name, middle initial)

HALLY, DAVID

6. DOCUMENT DATE AUGUST 1985	7a. TOTAL NO. OF PAGES 44	7b. NO. OF REFS 3
8a. PROJECT OR GRANT NO.	9a. ORIGINATOR'S DOCUMENT NUMBER(S) D.R.E.A. TECHNICAL COMMUNICATION 85/312	
8b. CONTRACT NO.	9b. OTHER DOCUMENT NO.(S) (Any other numbers that may be assigned this document)	

10. DISTRIBUTION STATEMENT

11. SUPPLEMENTARY NOTES	12. SPONSORING ACTIVITY

13. ABSTRACT

The GETWRD Package is a library of procedures designed to ease the implementation of command languages using FORTRAN 77. It allows the user to interpret a word of user input by matching it with one of the entries in a user-supplied dictionary. Features designed to increase the friendliness of the program/user interface include a type ahead facility, recognition of abbreviations, word completion and understandable error messages.

This memorandum describes enhancements to the GETWRD Package. While most of the enhancements are of most benefit to the programmer, a major improvement to the program/user interface is the inclusion of a spelling corrector which will catch most typographical errors committed by the user. Other improvements include the ability to interpret the work of input as a simple string with no dictionary matching, a logical variable, the answer to a Yes-No question, or a number in a specified range; greater flexibility for the programmer in formatting prompt and help messages; and a sorting routine which can be used to ensure that the dictionary is in alphabetic order.

All changes have been implemented to be upwardly compatible with the original version of the GETWRD Package so that no changes need be made to existing code which calls GETWRD Package procedures.

DD FORM 1473
1 JAN 79

KEY WORDS

command language
program/user interface
typographical error correction
input procedures

INSTRUCTIONS

1. ORIGINATING ACTIVITY: Enter the name and address of the organization issuing the document.

2a. DOCUMENT SECURITY CLASSIFICATION. Enter the overall security classification of the document including special warning terms whenever applicable.

2b. GROUP: Enter security reclassification group number. The three groups are defined in Appendix 'M' of the DRB Security Regulations.

3. DOCUMENT TITLE: Enter the complete document title in all capital letters. Titles in all cases should be unclassified. If a sufficiently descriptive title cannot be selected without classification, show title classification with the usual one-capital-letter abbreviation in parentheses immediately following the title.

4. DESCRIPTIVE NOTES. Enter the category of document, e.g. technical report, technical note or technical letter. If appropriate, enter the type of document, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.

5. AUTHOR(S): Enter the name(s) of author(s) as shown on or in the document. Enter last name, first name, middle initial. If military, show rank. The name of the principal author is an absolute minimum requirement.

6. DOCUMENT DATE: Enter the date (month, year) of Establishment approval for publication of the document.

7a. TOTAL NUMBER OF PAGES: The total page count should follow normal pagination procedures, i.e., enter the number of pages containing information.

7b. NUMBER OF REFERENCES. Enter the total number of references cited in the document.

8a. PROJECT OR GRANT NUMBER: If appropriate, enter the applicable research and development project or grant number under which the document was written.

8b. CONTRACT NUMBER: If appropriate, enter the applicable number under which the document was written.

9a. ORIGINATOR'S DOCUMENT NUMBER(S). Enter the official document number by which the document will be identified and controlled by the originating activity. This number must be unique to this document.

9b. OTHER DOCUMENT NUMBER(S): If the document has been assigned any other document numbers (either by the originator or by the sponsor), also enter this number(s).

10. DISTRIBUTION STATEMENT: Enter any limitations on further dissemination of the document, other than those imposed by security classification, using standard statements such as:

(1) "Qualified requesters may obtain copies of this document from their defence documentation center."

(2) "Announcement and dissemination of this document is not authorized without prior approval from originating activity."

11. SUPPLEMENTARY NOTES: Use for additional explanatory notes.

12. SPONSORING ACTIVITY: Enter the name of the departmental project office or laboratory sponsoring the research and development. Include address.

13. ABSTRACT: Enter an abstract giving a brief and factual summary of the document, even though it may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall end with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (TS), (S), (C), (R), or (U).

The length of the abstract should be limited to 20 single-spaced standard typewritten lines; 7½ inches long.

14. KEY WORDS: Key words are technically meaningful terms or short phrases that characterize a document and could be helpful in cataloging the document. Key words should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location, may be used as key words but will be followed by an indication of technical context.

END

FILMED

1-86

DTIC